The Analysis of Parallelism of Apache Storm

Xinyao Zhang

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Electrical and Computer Engineering) at

Concordia University

Montreal, Canada

June 2016

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Xinyao Zhang

Entitled: The Analysis of Parallelism of Apache Storm

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. Rabin Raut Chair

Dr. Lingyu Wang Examiner

Dr. Anjali Agarwal Examiner

Dr. Dongyu Qiu Supervisor

Approved by _____

Chair of Department or Graduate Program Director

_____

Dean of Faculty

Date August 30, 2016

ABSTRACT

The Analysis of Parallelism of Apache Storm

Xinyao Zhang

Big data processing is rapidly growing in recent years due to the immediate demanding of many applications. This growth compels industries to leverage scheduling in order to optimally allocate the resources to the big data streams which require data-driven big data analysis. Moreover, optimal scheduling of big data stream process should guarantee the QoS requirements of computing tasks. Execution time of tasks within the streams is specified as one of the most significant QoS factors.

In this paper, I will introduce the currently widely used stream processing framework Storm, a distributed real-time computation platform, and study the scheduling and execution strategies of big data stream processes within it. First, a queueing theory approach to the modeling of the streams as a collection of sequential and parallel tasks is proposed. It is assumed that heterogeneous threads are required to handle various big data tasks such as processing, storing and searching which may have quite general service time distributions. Then, with the proposed model, an optimization problem is defined to minimize the total number of resources required to serve the big data streams while guarantying the QoS requirements of their tasks. An algorithm is also proposed to mitigate the complexity order of the optimization problem. The objective of this research is to minimize the stream processing resources in terms of threads with constraints

over the task waiting time of the application tasks. I apply the proposed scheduling algorithm to Apache Storm to optimize the cloud resource requirements. The experiment results validate the analysis.

ACKNOWLEDGEMENTS

I would like to express my gratefulness to Dr. Dongyu Qiu for his continues guidance and help throughout this research. I would also like to thank my family for their support.

Xinyao Zhang

Contents

List of Figures

List of Symbols

R	Parallelism level of a task
$c_{i,j}$	Number of processes dedicated to type j parallel tasks of bag i
T_R	Service time of a task with R folks
μ	Average service time of a task
$\eta_{R,i}$	Standard deviation of the service time
λ	Arrival rate of tasks
V	Vacation time of a thread
W	Waiting time of a task in the system
$\beta_{i,j}$	Number of type j parallel threads at i^{th} bag of tasks
Γ	Gamma function
α_r	Probability to execute the allocated task
I	Number of sequential bags of tasks
K	Normalization factor

D	QoS constraint
Λ_i	Lagrangian coefficient
V	Service time of the shared task

List of Abbreviations

IT	Information Technology
IDC	International Data Corporation
RDBMS	Relational Database Management System
HDFS	Hadoop Distributed File System
GFS	Google File System
SLA	Service Level Agreement
QoS	Quality of Service
JAR	Java Archive
JVM	Java Virtual Machine
API	Application Programming Interface
OOM	Out Of Memory

Chapter 1

Introduction

1.1 Introduction

Cloud computing has already led a great revolution in traditional Information Technology (IT) industry, it helps developers and companies overcome the lack of capacity in hardware (e.g. CPU and storage) and allows users indicate resources through the Internet in an on-demand fashion [0][2]. Meanwhile, big data concept grows in an extravagant rate due to the development of cloud-based video services like Netflix, social networks such as Facebook and file sharing applications including Dropbox, all these applications interact with the users frequently. According to the 2011 International Data Corporation (IDC) report, the total produced and copied data all around the world was 1.8ZB, which increased by nearly nine times in five years. And according the research of [3], it is predicted that, in the near future, this amount will be doubled every two years.

The massive volume of data processing, for instance, Facebook collect and analyze more than 700TB data set daily, makes traditional database with the strategy of decision-support no longer adequate at this time because they cannot scale well with the massive computing and storage resources [4].

1.1.1 Big Data

Big data is a widely used but often being misunderstood concept. There is no common definition of big data which is universal agreed, but in general, it shall presents massive data sets that in a tolerable time could not be captured, managed, or processed by traditional processing applications.

With the definition which Apache Hadoop made in 2010 that big data is "datasets which could not be captured, managed, and processed by general computers within an acceptable scope", McKinsey Global Institute, a global consulting agency, announced big data as the next frontier for innovation, competition, and productivity in 2011 [5]. From their definition, big data means the dataset that could not be acquired, stored, or managed by traditional database software due to their inadequate of capability. This definition includes two important features: First, the volumes of big data are changing, and will growing rapidly over time or with the development of technology. Second, in different applications the volume of big data should be different from each other. At present, big data generally ranges from TB to PB, however, from the definition given above, it is showed that the dataset volume is not the only important factor for big data, the increasingly growing data scale and its characteristic that cannot be handled by traditional database are two critical key features.

In fact, as early as 2001, in [6] and related researches, the data growth challenges and opportunities were given and defined as being three-dimensional, which are volume, velocity, and variety. These three dimensions (referred to 3Vs commonly) can be described as follows:

1. **Volume** refers to the amount of the data. The massive volumes of data that researchers and industries currently dealing with has required considering new storage and processing strategies in order to develop the tools which is needed to properly analyze it.
2. **Velocity** addresses the speed at which the data is generated and processed to meet the demands and challenges that lie in the path of growth and development. And it should be mentioned that, the velocity between batch processing, which works on historical data, and stream processing, which analyzes the data in real-time as it is generated, has several differences. This also implies to the rate of change of data, which is especially relevant in the area of stream processing.

3. **Variety** presents the issue of disparate and incompatible data formats. It indicates the various types of data, which include semi-structured and unstructured data such as audio, video, webpage, and text, as well as traditional structured data. Data can be retrieved from many different sources and take on many different forms, and it may take a significant amount of time and effort just in the preparation of analysis.

In 2012, the definition was updated to "Big data is high volume, high velocity, and/or high variety information assets that require new forms of processing to enable enhanced decision making, insight discovery and process optimization." And from [7], a new V "Veracity" should be extended to describe the big data, which presents the quality of captured data can vary great, which may affect accurate analysis.

However, compared to the 3V model, in 2011, International Data Corporation (IDC), one of the most influential research organizations in big data proposed that "big data technologies describe a new generation of technologies and architectures, designed to economically extract value from very large volumes of a wide variety of data, by enabling the high-velocity capture, discovery, and/or analysis" [8]. And this definition summarized the characteristics of big data as a 4V model, i.e., Volume, Variety, Velocity and Value. The fourth V "Value" points out the meaning and necessity of big data is exploring the huge hidden values from datasets with an enormous scale, various types, and rapid generation, which makes the 4Vs model widely recognized.

The definition of big data has been discussed fiercely from both academia and industry for a long time. In [9], the existing definitions of big data were classified into three categories: Attribute Definition, Comparative Definition and Architectural Definition. It also presented a big data technology map that illustrates its key technological evolutions.

1.1.2 Big Data Challenges

It was the first appearance of discussion about big data in modern computing in [10], although the research there was focused on data visualization, but the observations about the big data problem can easily be extended to general data analytics and machine learning. According to the paper, the big data problem consists of two distinct issues:

1. Big data datasets are the collection of datasets which could be processed individually; however, when they come together, for traditional hardware/software, the ability for

managing the collection would be inadequate. The data in the datasets could come from different sources, may be in various formats, and are stored in separate physical sites and in different types of repositories.

2. Big data objects are individual datasets which are too large to be processed by standard algorithms on available hardware and normally, they come from a single source.

The rapidly growing volume on data around the world these days brings huge challenges and opportunities on the area about data retrieving, storage, management and analysis. Relational database management system (RDBMS) is the one which traditional data process and analysis systems are based on. However, such RDBMSs only apply to the structured data, other than the case in big data which commonly is considered as unstructured data, which makes it apparently that the traditional RDBMSs are inadequate in handling the huge volume of big data. And for solving this problem, a lot of researches has been done to accomplish the achievement, which motive the appearance of cloud computing. In order to solve permanent storage and management of large-scale disordered datasets problems, distributed file systems like Hadoop Distributed File System (HDFS), Google File System (GFS) and NoSQL databases such as MongoDB and Apache River are good choices. Such programming frameworks have achieved great success in processing clustered tasks.

Before developing big data applications, there are several key challenges should be considered first:

1. **Data representation**: the difference in the type, structure and accessibility of datasets make it more meaningful to have a better data representation in computer analysis and user explanation. Efficient data representation shall reflect data structure, class, and type which make efficient operation possible. However, on the other hand, a chaos presentation of data makes it difficult and increase the cost in processing and analyzing.
2. **Redundancy reduction and data compression**: normally, when raw data comes into the system, it always comes with the high level of redundancy [11]. It would be pretty efficient if that is possible to reduce the redundancy or make a better compression of metadata.
3. **Data life cycle management**: the big data generating rate is overpowered when it compares to the traditional storage system ability, it makes the current storage system

could not afford such massive data. Therefore, it would be important to decide which data shall be stored and which data shall be discarded to make the data storage valuable.

4. **Analytical mechanism**: the time for big data system to process the analysis is limited. However, RDBMSs are designed with a lack of scalability and expandability, which could not meet the performance requirements. NoSQL databases show their advantages in the situation of unstructured data and become main storage system in big data analysis. Even with the help of NoSQL, there are still some problems about performance and particular applications. To solve the problem, some enterprises have utilized a mixed database architecture that integrates the advantages of both types. More researches are focusing on the in-memory database and sample data based on approximate analysis.
5. **Data confidentiality**: most big data service providers at present could not manage or analyze such huge datasets on their own because of the limited capacity of unique platform. It is considerable to do the processing with the help of professional tools. And in this case, the transmission of big data to a third party may cause the concern about data safety and security.
6. **Energy management**: the energy consumption of processing the data has always being the important fact which drawn much attention from both economy and environment perspectives. With the increment of data volume and analytical demands, the processing, storage, and transmission of big data will inevitably consume more and more electric energy. Therefore, system-level power consumption control and management mechanism shall be established for big data while the expandability and accessibility are ensured.
7. **Expendability and scalability**: it is important to make sure that the produced big data system could meet the requirement not only for the current but also the future. The algorithm should have the ability in processing increasingly expanding and more complex datasets.
8. **Cooperation**: big data is an interdisciplinary research, which requires experts in different fields cooperate together to explore the potential of big data. Therefore, it is important to establish a comprehensive network for scientists and engineers from different fields to work together, which makes it efficient to accomplish the analysis problem.

1.1.3 Hadoop

Under this background, many Internet companies such as Google, Facebook and Twitter, which are all relies on the ability of processing large volume of data to drive their core business, have already applied their own big data processing framework.

There are several big data processing platforms, however, when it comes to big data, it is Hadoop/MapReduce comes to the sight of people first.

Hadoop was initially introduced in 2007 as an open source implementation of the MapReduce processing engine linked with a distributed file system. It is a framework for running applications on cluster. The Hadoop framework provides applications both reliability and data motion. Map/Reduce is the analysis strategy which Hadoop implemented in it, where the application is divided into many small fragments of work, each of which may be executed or re-executed on any node in the cluster. In addition, the provided distributed file system (HDFS) that stores data on the compute nodes, providing high aggregate bandwidth across the cluster. Both MapReduce and the Hadoop Distributed File System are designed to ensure node failures can be automatically handled by the framework [12].

1.1.4 Storm

Processing high volumes of data collected over a short period of time requires efficient, scalable data storage and computing platform. The typical processing platforms such as Hadoop are designed for batch-oriented data processing paradigm which requires separate programs for input, process and output. A batch processing system contains Map/Reduce and is generally more concerned with size and complexity of jobs than latency of computation. However, exponential growth of real-time applications in a large scale, where computations can be divided into independent tasks, urges a new computing paradigm called real-time stream processing platform. This type of platform is doing for real-time processing what Hadoop did for batch processing, facilitating the unbounded real-time stream processing of data. Scalability, fault-tolerance and data guarantee are the desired characteristics of this computing platform. Management capability of the computing nodes is another key factor in performance evaluation of the real-time stream processing platforms.

Despite this surge of interest in real-time stream computing, in comparison with batch processing, real-time stream processing platforms are relatively unfledged. Comparing to Hadoop, the most successful batch processing data intensive computing framework, Storm is a real-time distributed stream data processing engine at Twitter that powers the real-time stream data management tasks which are crucial to provide Twitter services [13]. Storm--developed under the Apache License--is the pioneers of real-time stream processing systems, which is vastly applied in big data solutions. For instance, Twitter employs Storm to do the scalable processing in real-time in order to deal with over 500 million Tweets per day.

Apache Storm is a free and open source distributed real-time computation system. Storm makes it easy to reliably process unbounded streams of data, doing for real-time processing what Hadoop did for batch processing and it can be used with any programming language [14]. Storm is designed to be:

1. **Scalable**: It is important to make sure that the system is easy to add or remove nodes from a cluster without shutting down the whole topologies (which will be defined later). In Storm, in order to scale the topology, all the things that user need to do is to increase the number of resources (e.g. CPUs and Memory) into the topology and reset the parallelism setting.
2. **Fault-tolerant**: The ability of fault-tolerant is extremely important for storm as it processes massive data all time and should not be interrupted by a minimal failure, such as hardware fail in a nodes of the storm cluster. Storm can redeploy tasks when it is necessary.
3. **Data guarantee**: No data loss is one of essential requirements for a data processing system. The risk of losing data would not be accepted in the use of most fields, especially for those ask for accurate results. Storm makes sure that all the data would be processed as they are designed during their processing in the topology.
4. **Support of multiple programming languages**: Storm allows programmers to define and submit the topology in any languages as the core of Storm is defined by Thrift. Spouts and Bolts (the definition will be given later) are also able to be written in any programming languages.

The appearance of Storm fills the hole of data processing technology in real-time streaming system.

1.2 Motivation

For the propose of saving time and money and taking advantage of remote resources, it is necessary to use parallel computing in data-intensive computing. However, considering volume and veracity, one of the main concerns related to the big data in parallel computing system is access efficiency. Resources of the cloud should be minimized in retrieving (read or write into) files process, the delay also should meet the users Service Level Agreement (SLA) constraints. As the number of cloud resources (e.g. bandwidth and servers) increases access efficiency decrease. The processing of data intensive applications devote most of their execution time in disk I/O and the randomness of the delay for retrieving data from the data and applying it to process may cause violation over the SLA constraint.

Furthermore, cloud leverages limit the number of parallel threads to process tasks, but these threads may be failed due to the hardware or software problems. Hence, to avoid violating Quality of Service (QoS) constraints, data processing threads of QoS sensitive applications may be replicated. Thus, if software or hardware process corruption happens, the process of the running task will be continued in replicated threads.

And considering the mentioned problems, it would be necessary to decide the number of threads for the different tasks in the Storm topology. Fortunately, Storm provides the feature of parallelism in an easy way for the user to pre-configure it in the cluster, as long as the numbers of threads for each component (Spout and Bolt, which will be introduced later) are pointed out clearly before the deployment of the topology according to what have been mentioned before. Thus, fine-grained analysis of big data streams helps model and optimize the performance of the real-time stream processing.

1.3 Literature Review

Storm is a distributed real-time computation system build for intensive data processing, when the

computation model fit streaming process, it provide the best performance. There are a lot of works have been done in the streaming process in web applications. A model studied the overall throughput and buffer need for streaming application on heterogeneous hardware in [19], and [20] developed a model based on a network of queues, where the queues represent different tiers of the application.

With the development of cloud computing and big data technology, more and more research in parallel computing has drawn the attention of researchers. Vast number of papers addressed scalability and fault tolerance as the main objectives for designing and building stream processing applications. However, the objective of the research in this paper is to minimize the required resources of the stream was barely mentioned among those exist researches. One of the reference papers to model the stream processing is [21]. Based on the queuing network models, it is analyzed in [21] that the service time of tasks running on a parallel environment. Taking the same approach taken in [21], a few papers such as [22] and [23] introduced and developed some optimizations for stream processing. The modeling of the stream tasks proposed in this paper is similar to [21] and the proposed optimization problem stand on the shoulders of analysis of [22] and [23]. On the other hand, with the increasingly usage of Hadoop/MapReduce framework, several studies such as [24] describes a detailed set of mathematical performance models for describing the execution of a MapReduce job on Hadoop. A mathematically model based on closed queuing networks is proposed in [25], it predicts the execution time of the map phase of a MapReduce job. [26] presented a model-driven performance analysis approach for real-time streaming applications to pinpoint their controllable properties. Moreover, vast studies have been done on the cloud management platforms dedicated to low-latency processing. One of the main obstacles which increase the latency is retrieving the distributed file systems. Cloud system should leverage the number of parallel threads which are using in retrieving the chunks of files and processing the tasks of jobs.

It was showed in [15] to [18] that the time for retrieving the chunks from cloud and processing the data may be approximated as exponential distribution. It is reasonable to use multiple server queuing models with different arrival data retrieving requests which server is corresponded to threads with exponentially distributed service time, [15] modeled the latency of cloud storages data retrieving. There is a trade-off between number of threads to retrieve data and data

downloading latency. Therefore, for different class of users according to their Service Level Agreement (SLA), different number of threads should be assigned. In this paper, I dig into this problem to find the most efficiency number of threads for each type of processes for the use of access.

There are a lot of researches on stream processing that have been done in the web application context. [19] studied the overall throughput and buffer needed for streaming application on heterogeneous hardware. [20] developed a model based on a network of queues, where the queues represent different tiers of the application. The queuing modeling in this paper is close to the work in [20][21][25][32].

1.4 Contributions

In this paper, first the structure and data model of Storm will be introduced and discussed in details. Then, the task running strategies in parallel computing will be described in order to get the service time of stream tasks. After that, different models for stream processing and its associated required resources are proposed and analyzed by using the queuing theory approach. According to [15]–[18], the task response time of stream process is exponentially distributed. Therefore *G/M/c* queues are used to model the performance of the threads in the data process. The analysis is then extended to systems with general arrival and service time distributions. Based on the proposed model, with the objective of minimizing the number of resources required to serve the stream, an optimization problem is defined. Finally, a heuristic algorithm is proposed to mitigate the complexity of the optimization problem.

1.5 Thesis Organization

This paper is organized as follows.

Chapter 1 *Introduction* presents the introduction and motivation which shows the necessity to analysis real-time big data stream processing procedure in the Strom.

Chapter 2 *Storm Overview* gives the structure of the Storm, it analysis Storm in its architecture, stream working flow, grouping strategies, parallelism and the message guarantee method.

Chapter 3 *Modeling of the Storm* illustrates the analytical model for the stream tasks, and then introduces the appropriate queuing models that could be used in analyzing the Storm real-time stream processing as well, in the same chapter, the optimization problem about minimize the resources of Storm is defined and solved in the end.

Chapter 4 *Experiment Results and Analysis* validates the accuracy and effectiveness of the proposed model and algorithm with the help of experiment result.

Chapter 5 *Conclusions and Future Work* provides the conclusion and future works about this thesis.

Chapter 2

Storm Overview

Storm is an open source distributed real-time computation system [13][14]. It is written in Clojure and Java. The cluster of Storm is Hadoop like, it fills the void of Hadoop/ MapReduce system which provide general framework for batch processing, and Storm provides a real-time stream processing framework.

2.1 Storm Architecture

Similarly like Hadoop runs "MapReduce jobs", Storm runs "topologies" to perform their computation. However, unlike a MapReduce job in Hadoop which will finish eventually, the processing of a topology will never end unless the operation team decides to kill it.

2.1.1 Nimbus

There are two kinds of nodes, master node and worker node, working on a Storm cluster. The topologies are submitted to "Nimbus", a daemon running on the master node. "Nimbus" works similar like the "JobTracker" in Hadoop, which is been treated as the link between the user and

the Storm system. It is used for distributing code around the cluster, assigning tasks to machines and monitoring failures.

“Nimbus” is a set of Apache Thrift [27] implementation, and the topology definitions are Thrift objects with the combination of uploading a JAR (Java archive) file contains a class which builds a topology by using TopologyBuilder class. With the design of sending the Thrift object topology to “Nimbus”, the Storm topology can be written in any programming language. During the processing, the Nimbus tracks the topology and contacts with Supervisors using a heartbeat interval running on every nodes of the cluster to make sure that the topology is executing and figures out if there are any free resources. When starting a new topology on a Storm cluster, Nimbus schedules the tasks to free resources, and if there is no vacancy node available, it assigns the tasks to the most lightly workers. Once a Supervisor or one task is down, Nimbus will reassigns all the tasks running on the Supervisor’s worker or the task itself to other workers. And when there are available resources appear in the cluster, it rebalances the tasks running on the cluster to workers.

However, once the Nimbus is down, there is no way for users to submit new topologies to the cluster, and it cannot reassign the tasks which experience black out during this period unless the Nimbus is rebooted.

2.1.2 Supervisor

Each worker node runs a daemon called the "Supervisor". The “Supervisor” listens for work assigned to its machine and starts and stops worker processes as necessary based on what Nimbus has assigned to it [14]. It also keeps tracking the running situation of the workers and restarts them when they failed.

The “Supervisor” has three main events. The *heartbeat event* is scheduled to run in a predefined time interval. It monitors the health of the worker nodes, and reports if worker nodes are alive to the Nimbus. The *synchronize supervisor event* is responsible for managing the changes of the running topologies, including add new topologies into the Storm cluster and reassign exist tasks to free worker nodes. The *synchronize process* event manage the worker processes of the topology which running on node of the “Supervisor”. With the help of heartbeats, it records the workers situation as valid, time out, not started or disallowed. “Time out” state indicates that the

worker did not reply a heartbeat in the designed time range and should be assumed as dead nodes. “Not started” implies the worker is pending, either because of it belongs to a new submitted topology or move works which belong to an existing topology to the “Supervisor”. And the worker which has been noted as “disallowed” means it has been killed or being reassign to another node.

2.1.3 Zookeeper

Zookeeper [28] is responsible for the coordination between the “Nimbus” and the “Supervisor”. Figure 2.1 shows the communication between “Nimbus” and “Supervisors” through “Zookeeper”. The state of “Nimbus” and “Supervisor” and the configuration of Storm are kept in Zookeeper and local disk, this allows the work process forward even suffering a fail of “Nimbus” and restarts the workers if they are failed, this strategy provides incredible stable for the Storm cluster, and in this way, the “Nimbus” and “Supervisor” are both fail-fast and stateless.

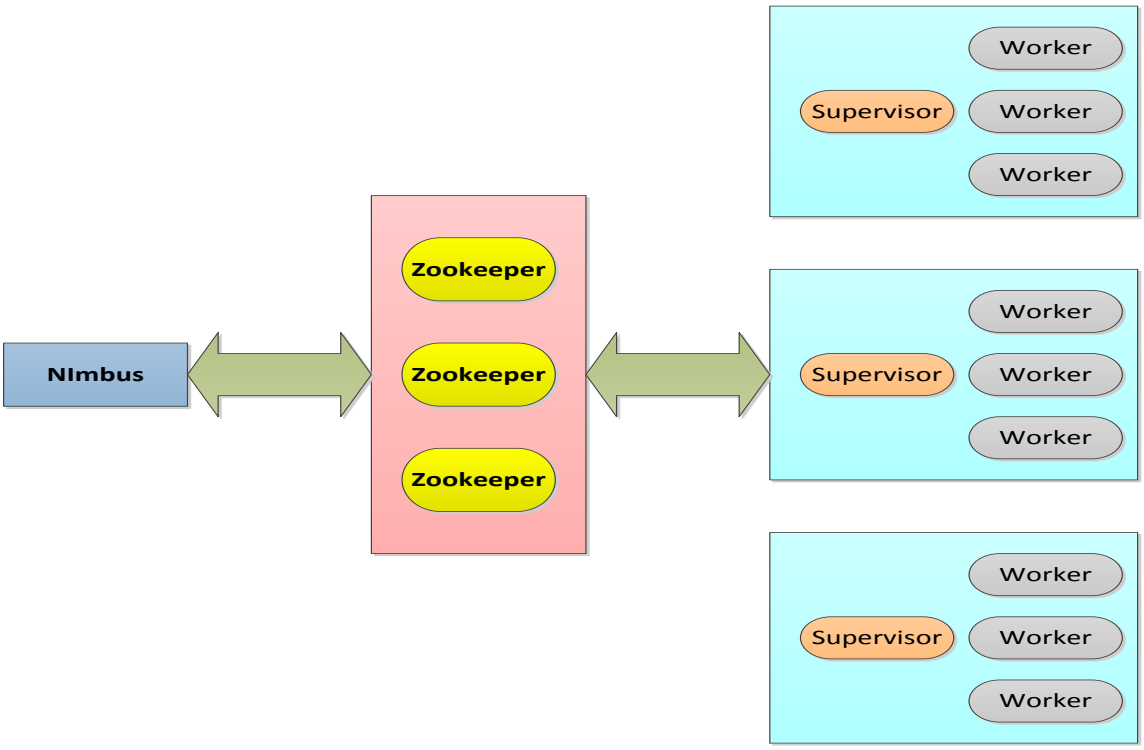

Figure 2.1: Architecture of Storm

2.1.4 Worker, Executor and Task

The actual work of the topology is done on worker nodes. For each worker node (machine), it may run one or more worker processes, and for each worker process, it will only run as part of a single topology which means there may have several worker processes running on different topologies in a single worker node, however, for each worker process it can only run on a single topology.

Each worker process runs a Java Virtual Machine (JVM), and several executors are running in it. Executor is thread running in the worker process, for each executor it may runs one or more same tasks in it. In Figure 2.2 we show a worker process on a worker node. This design is the key to implement the parallelism and scalable of the Storm framework, it will be described it briefly later. A task is the instance of a Spout or a Bolt, it does the actual data processing like query, logical computation and etc. The number of the tasks in a topology will be static after being assigned, however, the number of the executors can be modified dynamically during the running of the computation process.

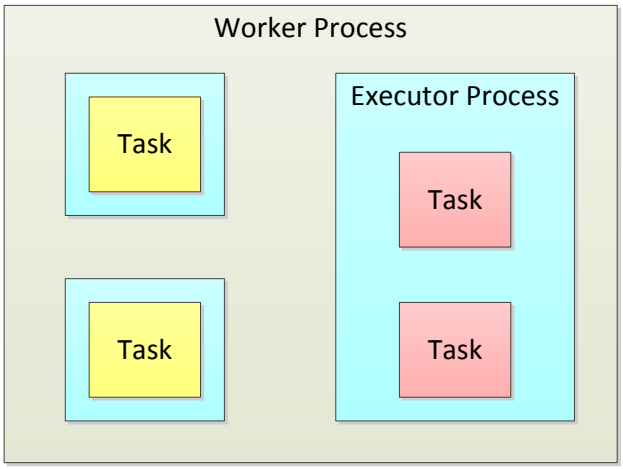

Figure 2.2: Worker Process

2.2 Stream

One topology is a directed graph of computation with the components of Spouts and Bolts. Each node (Spout or Bolt) in a topology performs the actual logical processing, and the edge between

nodes identifies the way how data should be transmitted between nodes.

Stream is the core abstraction of Storm, it is consisted of unbounded sequence of tuples. The tuple is a named list of value, and the field of a tuple can be the object of any type. Spout in the topology performs the source of the stream for the topology. And usually, it extracts data from extra source, like message queue such as Apache Kafak [29], Kestrel [30] or Application Programming Interface (API) of Twitter and then emits the streams. Bolt is the task that takes the input streams and do analytical processing then, perhaps, emits new stream to other Bolts which are linked to them. Complex stream computation which requires multiple steps need multiple Bolts. Bolt can do anything from run functions, filter tuples, streaming aggregations, streaming joins, talk to databases, and so on.

In Figure 2.3, one example of topology in Storm has been presented. Every nodes of the Storm topology executes in parallel. In the topology, the user can define the number of parallelism hint for each node, and Storm will spawn that number of threads across the cluster to do the execution. As for each executor in the worker process, there might be multiple tasks running inside of it, the task would support inter-topology parallelism and the executor would provide intra-topology parallelism. The means that how Storm deal with the parallelism shall be discussed later.

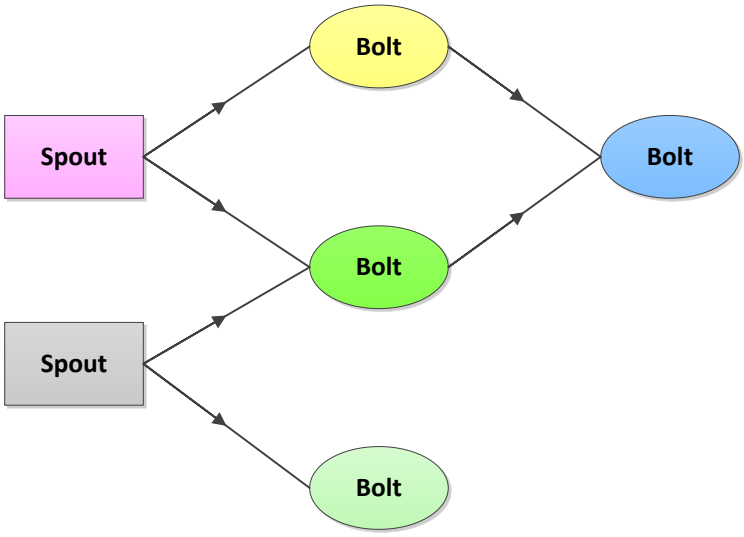

Figure 2.3: Storm Topology

2.3 Stream Grouping

As Spouts and Bolts contain multiple same tasks and they are running in parallelism way in the Storm cluster, the Storm using different stream grouping strategies to decide how to send tuples between tasks. The shuffling strategies of the stream are shown in Figure 2.4, it indicates the way how a topology execute at task level.

The following strategies are the shuffling methods which are provided by Storm:

1. **Shuffle grouping**: This is the simplest kind of steaming strategy, it sends the tuple to every task with a same probability.
2. **Field grouping**: It groups the stream by key value which is assigned to the tuple, by using this kind of grouping strategy, the tuple with same property could be shuffled together.
3. **All grouping**: It replicates and sends all the tuples to every task in the downstream.
4. **Global grouping**: It sends all the tuples to one and only one task.
5. **None grouping**: In this strategy, it does not care about where to send the tuple, and currently, it performs the same as shuffle grouping.
6. **Direct grouping**: This will let the user to decide which task will be responsible for receiving the tuple.

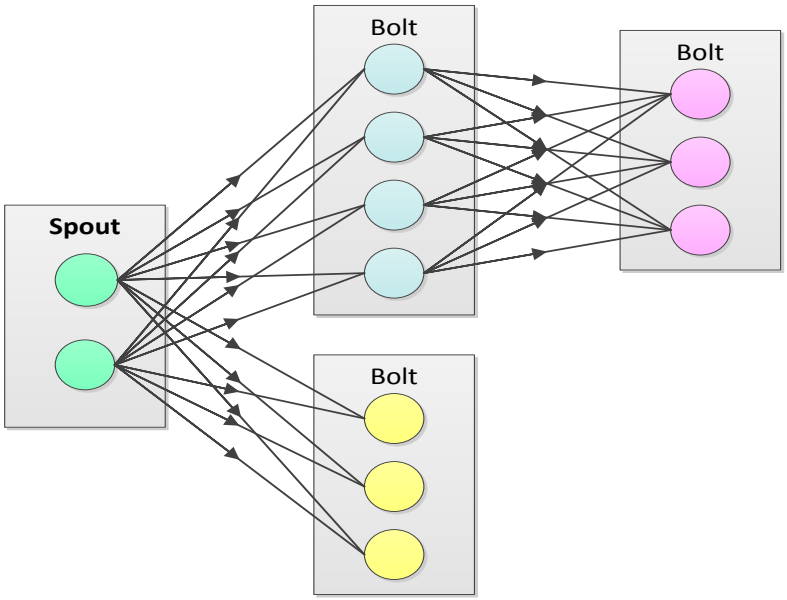

Figure 2.4: Shuffling of the Stream

2.4 Parallelism of Storm Topology

Remember that in a worker process there may have multiple executors, and in each executor it can contains multiple tasks. The parallelism of Storm is been supported in this design.

It is shown in [31] that how Storm will arrange the parallelism dynamically. Storm configures multiple tasks in an executor (thread), and as there is only one thread for multiple Spout or Bolt instances, it can be running in sequence only. However, with the design of the number of executor can be changed during the execution procedure, when there are more resources available, the Storm could use the "*storm rebalance*" command to expand the topology without taking the topology offline. This provides the flexibility of the parallelism for Storm topology.

Like Figure 2.5 shows, the parallelism hint of Spout A, Bolt B and Bolt C is 2, 2, 6 respectively. The total parallelism hint is 10 and the components are been spawned into 2 worker process, which means for each worker process there will be 5 threads running inside of it. After submitting and starting the topology, if there are more resources (e.g. new machine or released node) become available for the worker process, Storm could reassign the number of Bolt B executor into 2, which means that the Bolt B instances can run in parallelism instead of sequentially in that worker process.

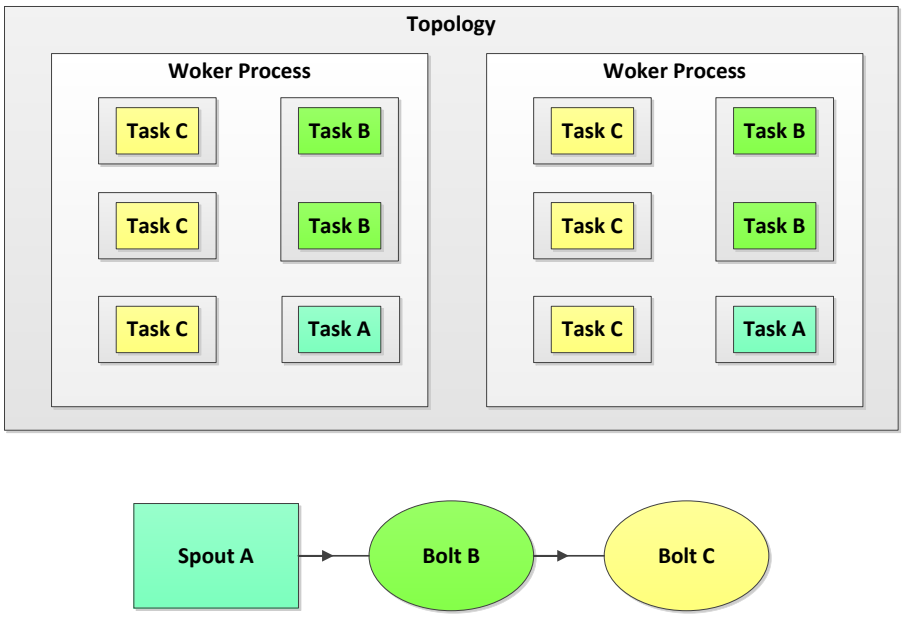

Figure 2.5: Storm Parallelism

2.5 Message Processing Guarantee

For all the big data analysis applications, one critical factor for them is to provide message guarantee strategies. And from the beginning, Storm is designed to guarantee that each message coming off from a Spout will be fully processed.

When processing a tuple which is emitted by a Spout, it may create new tuples by Bolts during the execution procedure, and all the tuples which are produced from the tuple before it leaves the system consist a tuple tree. Storm decides a tuple coming off a Spout as fully processed when the tuple tree of the original tuple has been exhausted and every message in the tree has been executed. A tuple will be considered failed when either one tuple or multiple tuples from the tuple tree could not be fully processed within a specified timeout.

To make sure that all the tuples will be fully processed, Storm uses the bitwise XORs methods to achieve the goal. When one tuple enters the Spout, a 64-bit message id will be attached to it which will be used to identify it, and after processing, it may emit one or multiple tuples, then for each downstream produced tuples, another random 64-bit id number will also be given. Every tuple records the id of its original Spout tuple which it exists in their tuple trees. When transmitting a new tuple into a Bolt, the Spout tuple id from the ancestor of the tuple will be passed into the new tuple as well. Next, the tuple will be sent to Bolts and Storm will take care of tracking the tuple tree. Once the tuple is fully processed, Storm will call the ack method on the Spout task which sends the original tuple. However, if the tuple suffers a times-out, Storm will call the fail method on the Spout.

When one message is sent into a Spout in the topology, the message will be marked as “pending”. A pending state indicates that the message is not ready to be taken off from the queue. And in the pending state, the message will not be able to be sent to other consumers. Then if the Spout call ack function, the pending message will be taken off from the queue, however, while a fail task is called, the message will be retransmitted into the Spout component.

A specific acker Bolt will be used to track all the Spout tuple like the one which is shown in Figure 2.6. When a tuple is full processed, the acker who tracks the tuple will send a message to the Spout to ack the message. User can configure the number of ackers in the topology to provide

better processing. As the acker will track every tuples in the tuple tree, it may cause OOM (Out Of Memory) issue, the ackers will take a strategy that only requires a fixed amount of space per Spout tuple (about 20 bytes).

An acker task stores a map from a Spout tuple id to a pair of values. The first value is the task id that created by the Spout tuple which will be used later to send completion messages. The second value is a 64 bit number called the "ack value". The ack value is a representation of the state of the entire tuple tree, no matter how big or how small.

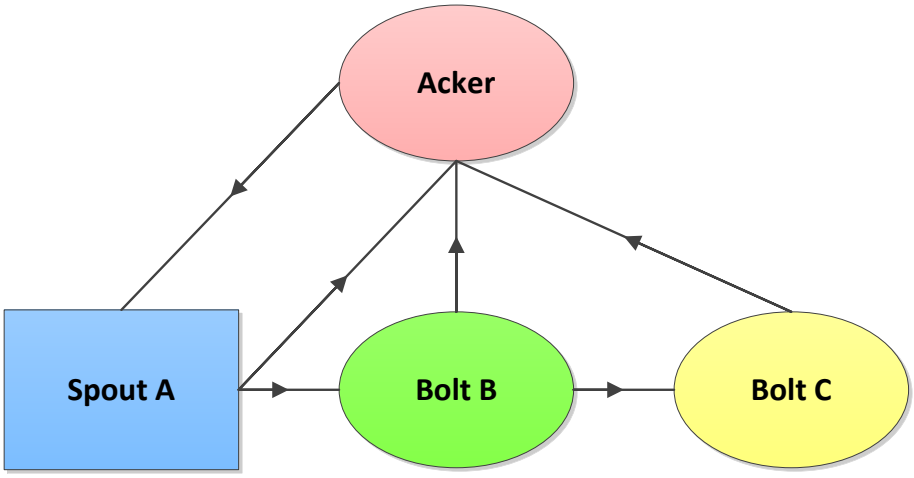

Figure 2.6: Message Guarantee

For each tuple in the tuple tree, there is a unique 64 bit number which will be assigned to it. When new tuples are been produced, they are XORed as the ack value and sent to the acker Bolt with the original tuple message id. When a tuple processed completely or acked, its message id and its original tuple message id will be sent to the acker Bolt. This acked tuple id is again XORed with the ack value. If the ack value goes to zero, it means that the tuple is fully processed and the acker Bolt sends the final ack to the Spout in order to admit the tuple.

When an acker task sees that the ack value has become 0, it knows that the tuple tree is completed. Since tuple ids are random 64 bit numbers, the chances of an ack value accidentally becoming 0 is extremely small. Working with the math, at 10K acks per second, it will take

50,000,000 years until a mistake is made. And even then, it will only cause data loss if that tuple happens to fail in the topology [14].

Chapter 3

Modeling of the Storm

In order to optimize the Storm cluster resources, it is critical to analysis the system first. In this section, the model which will be used in analyzing streaming processing will be introduced. Secondly, the tasks running strategies will be discussed as well, with modeling all those strategies, it makes it possible to calculate the service time of stream task in groups. These would be the fundament for calculating the tasks delay later. After that, according to the obtained service times, two queuing models for calculating the waiting time are proposed. Then, based on the proposed models, with the objective of minimizing the number of resources required to serve the stream, an optimization problem is defined. Finally, a heuristic algorithm is proposed to mitigate the complexity of the optimization problem.

3.1 Model of Stream Processing

Figure 3.1 shows the model that is used for the stream computing. In queueing network model, physical resources (processors, memory, network, etc.) will be considered as a service center. Modeling the physical resources in this way has been indicated as an appropriate method for

which could study the performance issue without including the detail of the real implementation [33].

A Storm topology can be modeled using the task graph, as shown in Figure 3.1. The nodes of the task graph represent the tasks of the parallel application and the edge of the graph link the nodes could be used to indicate the precedence constrains. A task corresponds to a series execution by a single processor and is specified by its service demands (or total mean service requirements) on the physical resources.

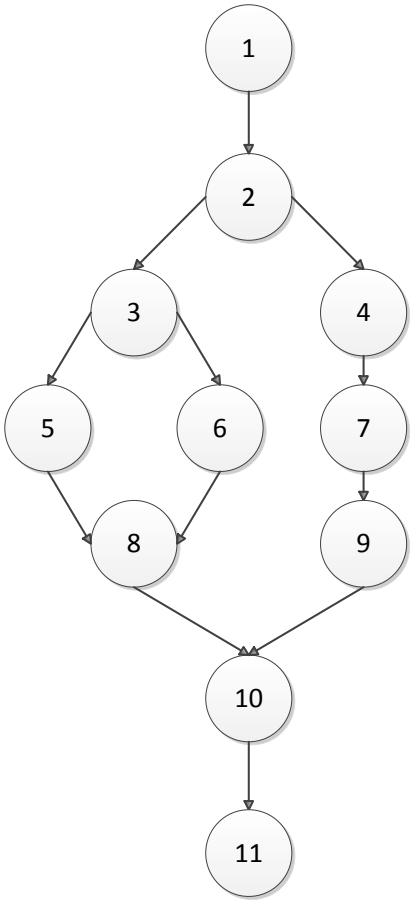

Figure 3.1: Storm Topology

This model is going to be used to study the system performance, and in this way, finding the result for our optimization result.

3.2 Model of Task Strategies

In order to analysis the performance of Storm topology, we need to build the task graph for it. As mentioned before, nodes represent tasks and edges express precedence constrains. Here the types of the task running strategies will be introduced.

From [34], they introduced an analytical model to predict the performance of parallel workloads, and as a reference, we modify that model to make sure that it could be used in Storm topology. In our model, we consider there would be four different types of execution method: Sequential (or type S), Parallel AND (P_1) Parallel OR (P_2) and Probabilistic Fork (P_3). Figure 3.2 (a) to (d) show how these task running strategies would be like in a task graph.

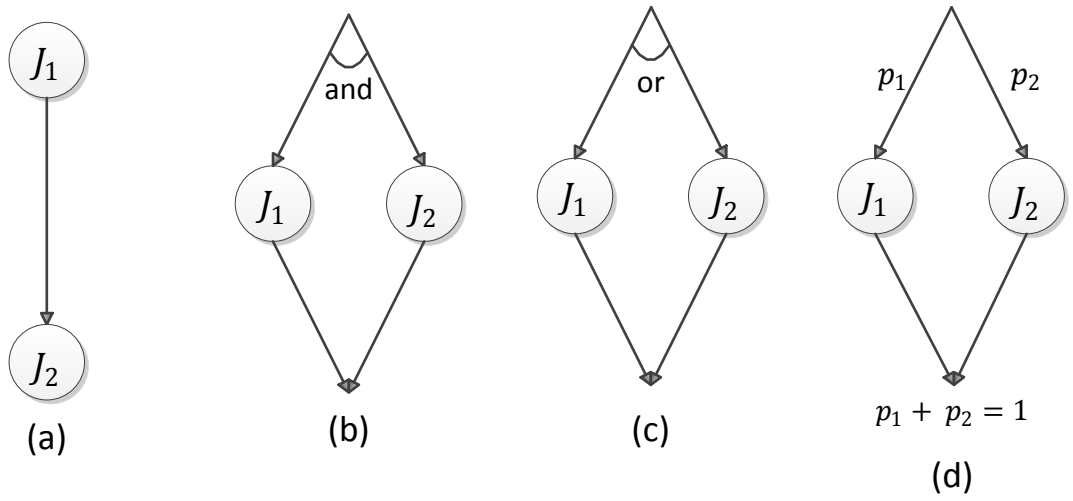

Figure 3.2: Types of task graph. (a) Sequential, (b) Parallel And, (c) Parallel Or, (d) Probabilistic Fork

As show in Figure 3.2 (a), a type S job indicates that job J consist of small jobs $J_1$and J_2 finish in sequence order, $J = J_1 + J_2$. A type P_1 job is a combination of small jobs in parallel, like in Figure 3.2 (b), the job J will not be finished until both job J_1 and job J_2 complete, $J = J_1 \wedge J_2$. And type P_2 job mean that the job after J will be able to continue only after J_1 or J_2 is complete like in Figure 3.2 (c), $J = J_1 \vee J_2$, furthermore, we consider that as soon as one of the jobs in a type P_2 job finishes, all other parallel jobs will be stopped. For a type P_3 job in Figure 3.2 (d), the job J

will only choose one fork of the task graph to execute, and after the completion of that task, job J would be considered as finished, $J = J_1 \setminus J_2$.

In the part below, the tasks delay will be discussed and calculated. The notation is shown in Table 3.1.

Variables	Indicator
R	Parallelism level of a task
$c_{i,j}$	Number of processes dedicated to type j parallel tasks of bag i
T_R	Service time of a task with R folks
μ	Average service time of a task
$\eta_{R,i}$	Standard deviation of the service time
λ	Arrival rate of tasks
V	Vacation time of a thread
W	Waiting time of a task in the system
$\beta_{i,j}$	Number of type j parallel threads at i^{th} bag of tasks

Table 3.1: Table of Notations

3.2.1 Parallel AND

As it is defined and showed in Figure 3.2 (b) before, Parallel AND type, P_1, is a combination of small tasks processing in parallel, the execution will not stop unless all tasks running in parallel are finished. Under this circumstance, let t_r denotes the service time of the r^{th} fork in the parallel computation, the service time of the Parallel AND with R forks, T_{R_1}, will be equal to the maximum service time of the parallel threads and can be written by,

$$T_{R_1} = \max(t_1, t_2, \dots, t_r \dots, t_R) \tag{3.1}$$

If the service time of R parallel threads are exponentially distributed with the same mean value. Then it is known as an i.i.d. distribution, which makes,

$$Pr(T_r < t) = \prod_{j=1}^{r} P(t_j < t) \tag{3.2}$$

From above, the corresponding pdf of service time will be,

$$P_{R_1}(t) = R\mu e^{-\mu t}(1 - e^{-\mu t})^{R-1} \tag{3.2}$$

where μ is the average service time of one task.

Then, the average service time can be calculated as follows,

$$\begin{aligned} \frac{1}{\mu_{R_1}} &= E\left[T_{R_1}\right] \\ &= \int_0^\infty t P_{R_1}(t)\, dt \\ &= \frac{1}{\mu}\sum_{r=1}^{R} \frac{\binom{R}{r}}{r} \end{aligned} \tag{3.3}$$

As it is seemed, with increase of R, the average service time enhances drastically. The Laplace Transform associated with the probability generating function (PGF) of the $P_{R_1}(t)$ distribution is given by [35],

$$T_{R_1}(s) = \frac{s\Gamma(r+1)\Gamma(s/\mu)}{\mu\Gamma(r+s/\mu+1)} \tag{3.4}$$

Γ represents the gamma function. Hence, the variance of the service time $\eta_{R_1}^2$ can be numerically calculated as follows,

$$\eta_{R_1}^2 = \frac{\partial^2 T_{R_1}(s)}{\partial^2 s}_{|s=0} - \left[\frac{\partial T_{R_1}(s)}{\partial^2 s}\right]^2_{|s=0} \tag{3.5}$$

Eq. (3.5) can be calculated numerically and then be used to find the upper bound of waiting delay.

3.2.2 Parallel OR

Parallel OR type, P_2, represents a type of execution that the stream can continue their procedure as soon as one of the tasks completes, furthermore, all the other tasks which continue running will be stopped. In this case, the service time of the parallel threads will be equal to the minimum service time of the threads as follows,

$$T_{R_2} = \min(t_1, t_2, \dots, t_r \dots, t_R) \tag{3.6}$$

If each thread is exponentially distributed with parameter μ, then, T_{R_2} will have exponential distribution with parameter $R\mu$. This gives the pdf of the Parallel OR service time,

$$P_{R_2}(t) = R\mu e^{-R\mu t} \tag{3.7}$$

And with the definition of exponential distribution, it would be easy to get the mean

$$\frac{1}{\mu_{R_2}} = E\left[T_{R_2}\right]$$

$$= \frac{1}{R\mu} \tag{3.8}$$

and the corresponded variance in this situation,

$$\eta_{R_2}^2 = \frac{1}{(R\mu)^2} \tag{3.9}$$

Substituting the service time characteristics in *G/M/c* model, it would be possible to calculate the task waiting delay.

3.2.3 Probabilistic Fork

For Probabilistic Fork type, P_3, the executioner will only choose one fork of the task graph with probability $\alpha_{3,r}$ to execute the allocated task. It should be mentioned that $\sum_{r=1}^{R} \alpha_{3,r} = 1$ in which again *R* represents the number of forks in parallel. According to [36], average service time is,

$$\frac{1}{\mu_{R_3}} = \sum_{r=1}^{R} \frac{\alpha_r}{\mu_r} \tag{3.10}$$

and similarly, the variance of the Probabilistic Fork type job is given by,

$$\eta_{R_3}^2 = \sum_{r=1}^{R} \frac{2\alpha_r - \alpha_r^2}{\mu_r^2} \tag{3.11}$$

3.2.4 Sequential

In almost all concurrent processing structures, group of tasks are sequentially dependent on other coupled groups of tasks such that when the coupled tasks are terminated, their output will be applied to initiate processing of the other tasks.

To simplify the model, like it is shown in Figure 3.3, it is assumed that each stream consists of I sequential bags of tasks so that each stream requires various number of threads. In each bag of tasks, it contains multiple all three types task executing strategy which for each type they share the same waiting delay in the same bag. Then the cumulative waiting delay of the stream can be defined as the accumulation of the delays resulted by the delay of these bags of tasks.

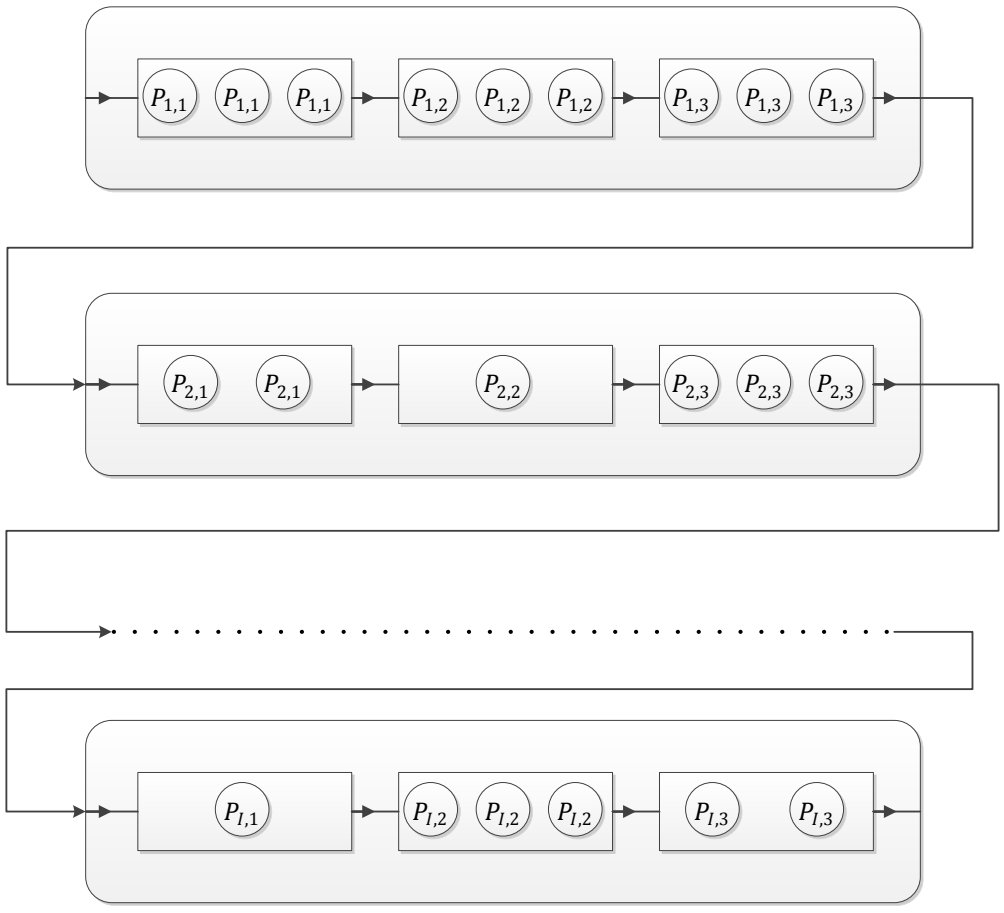

Figure 3.3: Sequential Packages

As it explained previously, we assume the different number of task execution types represented by J is equal to 3. Let us define $\beta_{i,j}$ as the number of type j parallel threads at bag of tasks i required to serve the stream. Then, total waiting delay W_T can be represented by,

$$W_T = \sum_{i=1}^{I} \sum_{j=1}^{J} \beta_{i,j} W_{i,j} \tag{3.12}$$

3.3 Model of System

As it is shown in Figure 3.4, multiple threads would be able to set to one task group. Threads run in the context of the process. So each process contains several threads. Let $c_{i,j}$ denote the number of processes required to serve type j tasks of bag i. Hence, if $R_{i,j}$ denote the number of threads allocated to the type j tasks of bag i, $W_{i,j}$ can be obtained from one of the following two *G/M/c* and *G/G/c* queuing models.

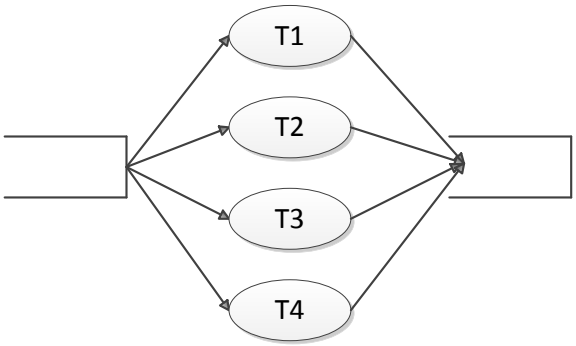

Figure 3.4: Stream Processing

3.3.1 *G/M/c* Model

G/*M*/*c* queue model can be applied to investigate the delay of the stream tasks in the system. Assuming general distribution for the task arrival rate, average waiting time $W_{i,j}$ of the tasks in the system is given by [36],

$$W_{i,j} = \frac{K\zeta^{c_{i,j}}}{c_{i,j}\zeta(1-\zeta)^2} \tag{3.13}$$

where *K* represents a normalization factor which can be obtained by letting sum of the queue length probabilities equal to 1. *c* denotes the number of executors(processes) to serve the tasks of stream and furthermore, assuming tasks arrive into the system according to the general distribution $A(t)$, ζcan be numerically calculated as follows,

$$\zeta = \int_0^\infty e^{-c_{i,j}\mu_{i,j}(1-\zeta)t}\partial A(t) \tag{3.14}$$

Then with the help of Eq. (3.3), Eq. (3.8) and Eq. (3.10), the upper bound of the waiting delay would be realistic to be found.

3.3.2 *G/G/c* Model

In the most complicated situation, when arrival rate and service time follow general distributions, the analysis has to be done with general assumption on arrival rate and service time. However due to the computational complexity, we will not be able to find the closed form waiting delay for this model. Hence, here we decide to use its upper bound. We define λ as the average task arrival rate, η_λ^2 as the variance of inter-arrival time and μ as the average service time of a task. η_μ^2 stands for the variance of the service time. Then, the upper bound of the waiting delay could be written by [37],

$$W_{i,j} \leq \frac{1}{\mu_{i,j}} + 2\lambda_{i,j}\frac{\eta_{\lambda_{i,j}}^2 + \frac{\eta_{\mu_{i,j}}^2}{c_{i,j}} + \frac{c_{i,j-1}}{\left(c_{i,j}\mu_{i,j}\right)^2}}{1-\frac{\lambda_{i,j}}{c_{i,j}\mu_{i,j}}} \tag{3.15}$$

Mean and standard deviation of service time different tasks, $\mu_{i,j}$ and $\eta_{\mu_{i.j}}$, calculated in previous subsection, from Eq. (3.3), Eq. (3.5), Eq. (3.8), Eq. (3.9), Eq. (3.10) and Eq. (3.11), should be applied into the upper-bound presented in Eq. (3.13) and Eq. (3.15) to obtain the waiting time delay of stream of type *j* tasks of bag *i*.

If the service time can be approximated highly enough by an exponential distribution, *G*/*M*/*c* model is preferable and give us a better approximation of the waiting time, if not, upper bound of *G*/*G*/*c* model should be applied.

3.4 Performance Optimization

Total number of threads in the system are equal to $\sum_{i=1}^{I}\sum_{j=1}^{J} R_{i,j}c_{i,j}$. Note that, for all i and j, $R_{i,j}$ depending on the task type is fixed and variables are $c_{i,j}$s. In this paper, the optimization objective is to minimize the total number of in-service threads considering the response delay constraint of streams. Thus, the optimization problem will be as follows,

$$\min_C \sum_{i=1}^{I}\sum_{j=1}^{J} R_{i,j}c_{i,j} \tag{3.16}$$

subject to

$$\sum_{j=1}^{J} \beta_{i,j}W_{i,j} \leq D_i, i = 1, \dots, I \tag{3.17}$$

As $c_{i,j}$s decrease the delay of associated stream will be augmented. Thus, to control the delay, number of threads attributed to the stream has a vital role. Due to the existence of the non-linear constraints, above optimization problem requires non-linear integer programming which is complicated and time consuming. Thus using the duality theory, the dual of the above optimization problem will be as follows,

$$\max_{C,\Lambda} \sum_{i=1}^{I}\sum_{j=1}^{J} R_{i,j}c_{i,j} + \sum_{i}^{I} \Lambda_i\left(\sum_{j}^{J} \beta_{i,j}W_{i,j} - D_i\right) \tag{3.18}$$

subject to $c_{i,j} \geq 0, i = 1, \dots, I, j = 1, \dots, J$

$$R_{i,j} + \sum_{i=1}^{I}\left(\Lambda_i \frac{\partial W_{i,j}}{\partial c_{i,j}}\right) = 0\,, i = 1, \dots, I, j = 1, \dots, J. \tag{3.19}$$

Where Λ_i is the Lagrangian coefficient associated with the delay constraint on the i^{th} bags of tasks. Considering the duality gap, relaxed solution can be found by solving the above optimization problem. Then, the typical way to find the final answer is to find the integer solution out of the relaxed solution using branch and bound or branch and cut algorithms.

However, after finding the relaxed $c_{i,j}$ coefficients, another policy is proposed to optimize the performance of the cloud data system. In this paper, we propose process sharing policy, so that a process is shared among the streams to serve their tasks. However, under these circumstances, the complexity order will be increased in a dramatic way.

Therefore, for non-integer $c_{i,j}$ values, Nearest Neighbor Search (NNS) is applied to find (i,j) and (i',j') pairs such that $c_{i,j} - [c_{i,j}] \approx 1 - (c_{i',j'} - [c_{i',j'}])$.

The low complexity order of NNS [38], $O(\log N)$ makes it the best candidate in this application. Thus, using NSS, pairs will be selected such that the summation of non-integer parts of $c_{i,j}$ and $c_{i',j'}$ would be equal to 1. Then a process will be shared between these two pairs according to the non-integer part of the relaxed optimal results so that the non-integer part dedicates the usage ratio of the process by each pair.

Algorithm 1: Thread Optimization

Initialization initialize $c = \{c_{1,1}, \dots, c_{i,j}, \dots, c_{I,J}\}$
Search and Find pairs (i,j) and (i',j') such that Argmax $c_{i',j'} - [c_{i',j'}] + c_{i,j} - [c_{i,j}] \leq 1$
ShareThread(); **while**
Running Tasks in the system **do**
for *Shared Threads* **do**
calculate the ratio of executed tasks of each shared stream;
if *The ratio is larger than the dedicated ratio && tasks of other stream arrives to the system* **then**
preempt the thread;
else
finish the job in a non-preemptive manner;
end
end
end

Dynamic sharing policy algorithm is represented in Algorithm 1. First, the number of tasks that are served by the shared process will be calculated. If the number of tasks of each job served is more than its ratio during the time window there will be two strategies namely non-preemptive and preemptive; in non-preemptive case, the task undergoing service is permitted to complete service without interruption even if the tasks of the other stream requires service in a meantime. Consequently, tasks of the partner stream may end up with longer delay which can be modeled

by a vacation time. Thus the delay for them has the extra part $\frac{\sigma_V^2}{\bar{V}}$ where V is the service time of the shared task. $\bar{V}$ and σ_V^2 represent its attributed mean and variance.

In preemptive case, service of a stream task passing its limit is interrupted when tasks of other shared stream arrives into the system. The extra delay for preempted case is for the stream violating its usage ratio and depends on the previous history of the tasks served by the shared process, and the maximum extra delay for the preempted task stream which previously has served much more served tasks than its allocated capacity will be equal to the gross processing time driven from the following equation,

$$E[p_g] = \frac{1-\varphi(\bar{\lambda}_V)}{\bar{\lambda}_V \varphi(\bar{\lambda}_V)} \tag{3.20}$$

where $\varphi(x)$ represents inverse Laplace transform of associated service time of preempted stream task. For more details, please check [39]. The complexity of process sharing management compels sharing policy to be restricted to no more than two pairs.

Chapter 4

Simulation and Experimental Results

In this section, a real Storm cluster will be built, and in the cluster, a topology will be processed. With the help of the setup environment, it would give the optimal result to prove that the proposed algorithm which is mentioned before would be valuable.

4.1 Experiment Environment

The cloud platform used to evaluate the proposed solution comprises two simple physical servers inter-connected by a Gigabit Ethernet switch. Each server is a Cisco UCS B200 M3 Blade Server System with two Intel Xeon Processor E5-2660 v2 CPUs and 8x16GB DDR3 (M393B2G70DB0-CMA) RAM. Each of the Xeon CPUs has 10 Cores and 25MB of L2 Cache and working in 2.2GHz frequency.

Also, each server has 500GB SCSCI 1500rpm HDD for storage. All servers run KVM hypervisor QEMU 2.0.0. Ubuntu 14.04 with Linux kernel 3.16 is chosen as the operating system. On top of the KVM hypervisor, Openstack Liberty with nova, cinder enabled in KVM VMs and heat and ceilometer enabled in another node called the controller node.

The Storm cluster is built with 3 nodes, one as the nimbus and others as supervisors. All the nodes are running Ubuntu 14.04.2 LTS with kernel 3.16.0-43-generic. For the node which acts like nimbus, it contains four Core i5-3210M 2.50GHz processors and 4.00GB RAM. Each of the two supervisors has eight Core i7 2.80GHz processors and 8.00GB RAM. Both of the supervisors run 8 worker slots at the same time. A worker process executes the subset of a topology, and runs in its own JVM. Each worker process can only be in one topology, but may run one or more executors for one or more components (Spouts or Bolts). One running topology contains many such processes on several nodes within a Storm cluster [31].

4.2 Topology Setup

For the experiment, there are two different topologies which are processed separately in the cluster, both of them run the typical WordCountTopology.

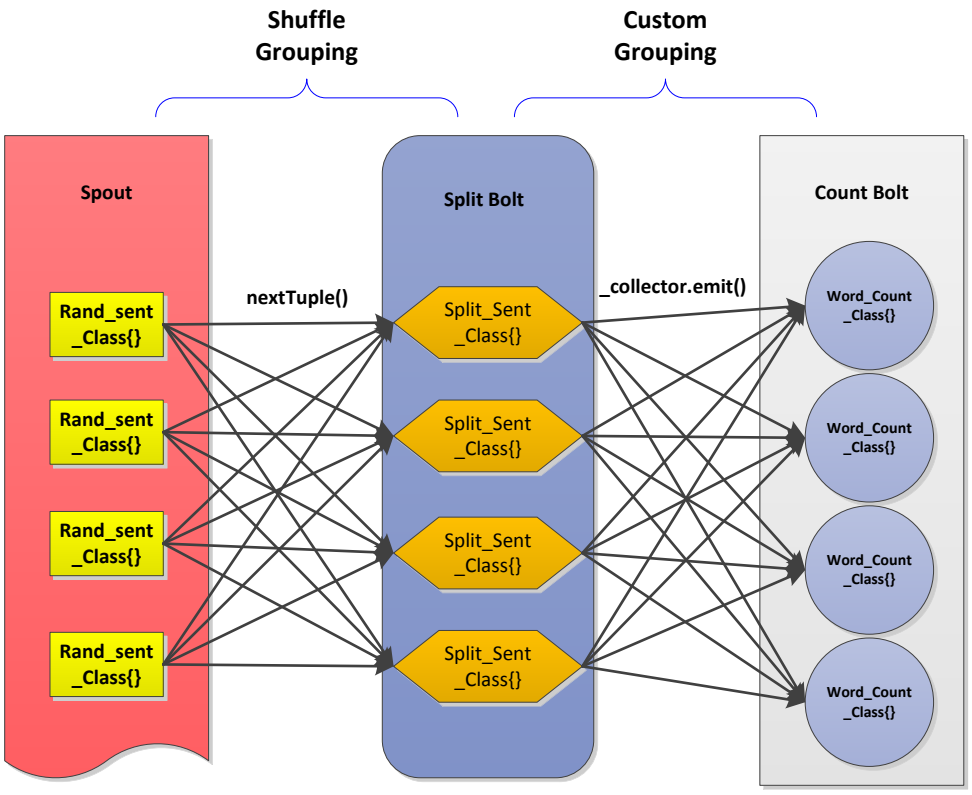

Figure 4.1: The First Scenario

For the first scenario, which is shown in Figure 4.1, a Spout contains four instances and each of them is processed in independent executors. An executor is a java thread that is spawned by a worker process and it may run one or more tasks for the same component (Spout or Bolt). An executor always has one thread that is used for all of its tasks, which means that tasks run serially on an executor [31]. All the Spout instances emit random sentence in shuffle grouping method to pass tuples to the split Bolt which contains four instances running in four different executors. Shuffle grouping method sends tuples to random tasks where tuples will be transmitted to the corresponding downstream Bolt instances with the same probability directly. After the processing of split Bolt, all the processed tuples will be transmitted to the next count Bolt, which as well, contains four tasks running in four executors. Unlike the way that tuples emit from Spout to split Bolt, the tuple passes from split Bolt to the count Bolt using the custom grouping; it makes sure that the same word produced by the split Bolt can be processed by the same count Bolt instances. Furthermore, the Bolt instance process word ordered by its initial letter.

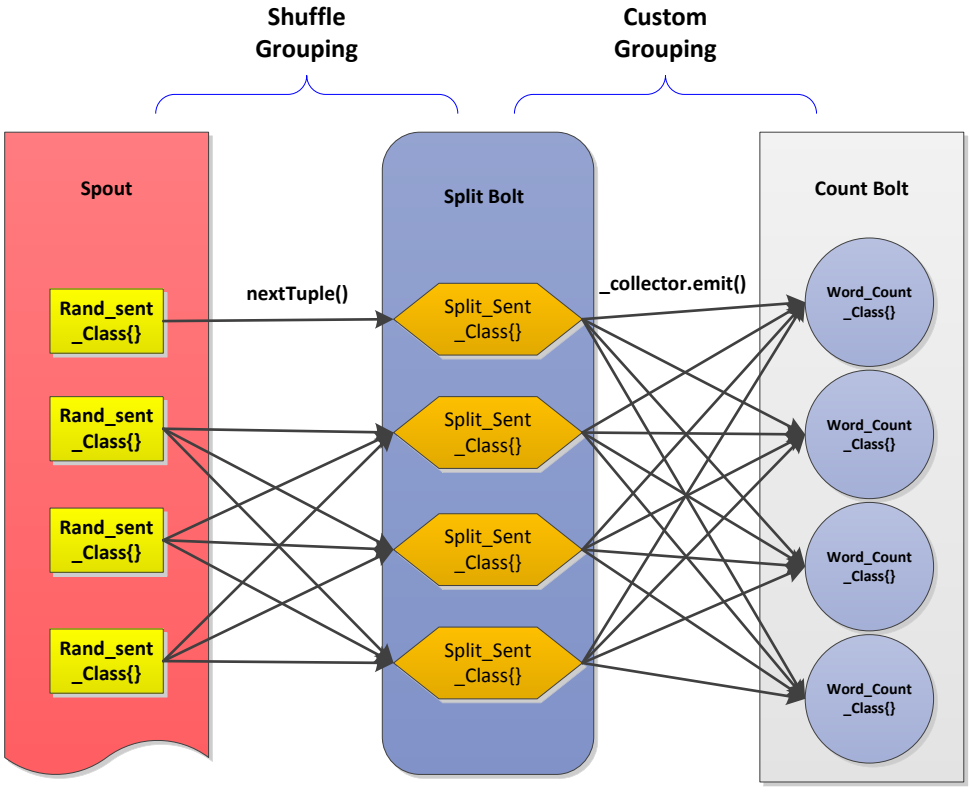

Figure 4.2: The Second Scenario

Then for the second scenario, one Spout and one split Bolt with four instances respectively will be set; however, in this topology, one of the Spout instances will be linked to a specific split Bolt instance: a dedicated channel is constructed between them, and the other three Spout tasks will be linked by fields grouping method to the left split Bolt tasks follow the first scenario. After the split Bolts processing, all the tuple produced by split tasks will be transmitted to count Bolt which contains four instances by using custom grouping method, like which is shown in Figure 4.2.

4.3 Experimental Results

In order to run the WordCountTopology, the data source should be got from the outside. The sentences are written and let them be emitted downstream in a random sequence; the other case is that the data are retrieved from twitter stream, and count the words that appear in the tweets.

After running the experiment, the service time of these scenarios is measured and represented in Figure 4.3. As it may seem, the service time distribution of the tasks follows the Gaussian Mixture Model rather than exponential distribution, this indicates that the *G/G/c* model in some complex scenarios should be applied rather than the *G/M/c*. The sentences are sent randomly such that the arrival rate does not follow any special distribution.

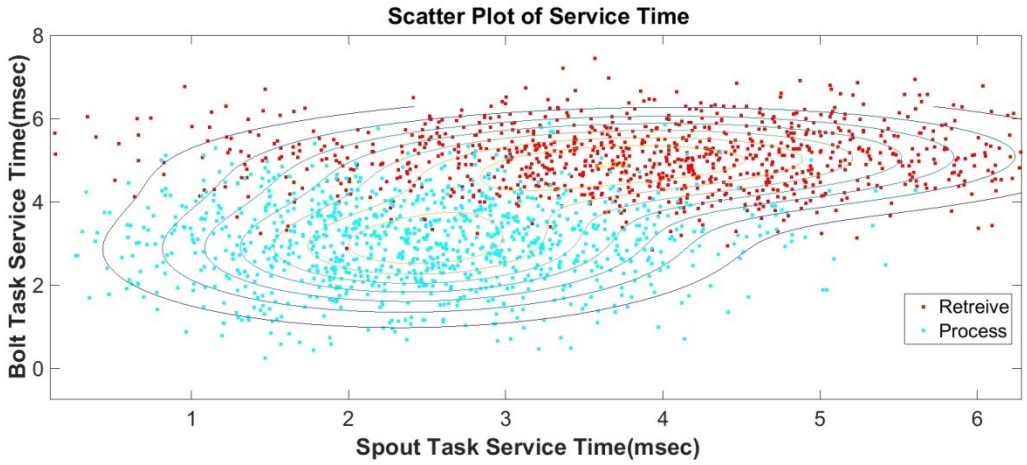

Figure 4.3: Word Count Scenario Service Time

In order to do the analysis, first, using the Maximum Likelihood, service time is approximated by an exponential distribution to be able to apply Eg. 4.9 for constraint over the delay. In parallel, the Eq. 4.11 is applied as the constraint over the delay while the service time follows general distribution.

Figure 4.4 shows the average stream delay as a function of total arrival sentence arrival rate. As it is dedicated, results by solving both equations lead to total delay less than the numerical *G*/*G*/*c* upper bound. Figure 4.4 indicates that either exponential approximation of the service time or upper-bound approximation of waiting time delay is both valid and countable in real scenarios.

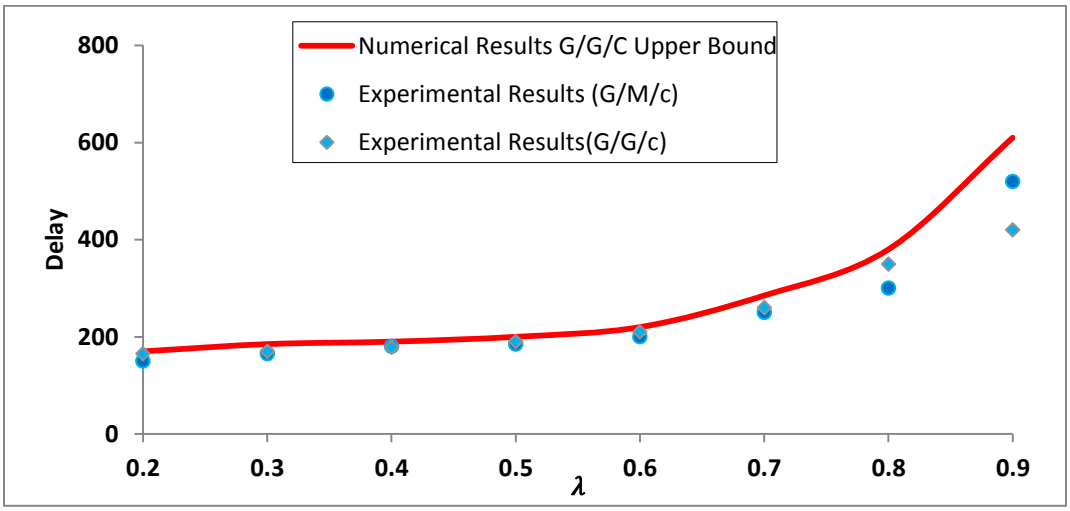

Figure 4.4: Function of Arrival Time

And it is also shown in Figure 4.5 that when the service time was controlled, both *G/G/c* and *G/M/c*, which means that no matter the task processing service time flows exponential distribution or just follow general distribution, the experimental waiting time of the stream will flow the upper bound of the numerical *G/G/c*, as same as the way when the arrival time changes.

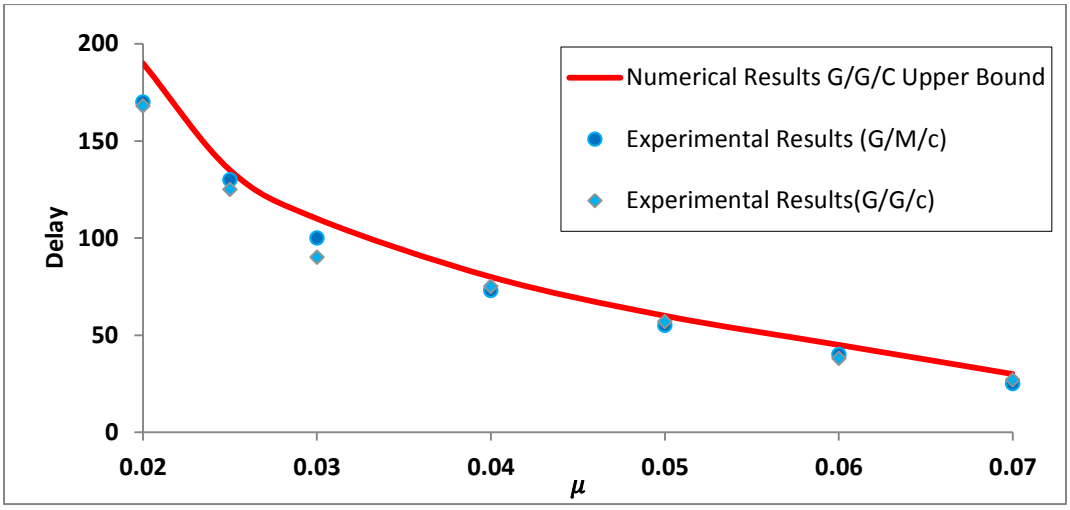

Figure 4.5: Function of Service Time

Last but not least, it should be mentioned that by having 10 streams in the system, 38 threads will be saved compared to the typical performance of the Storm nimbus. (With 5 streams, only 13 threads will be saved).

To better present the resource efficiency of the proposed optimized scheduler, in Figure 4.5 it is assumed that there are 10 Streams in the system. First, from processing the task in the dedicate threads considering the bound of the stream processing delay, number of threads required to finish the processing job are measured and presented in the yellow bar.

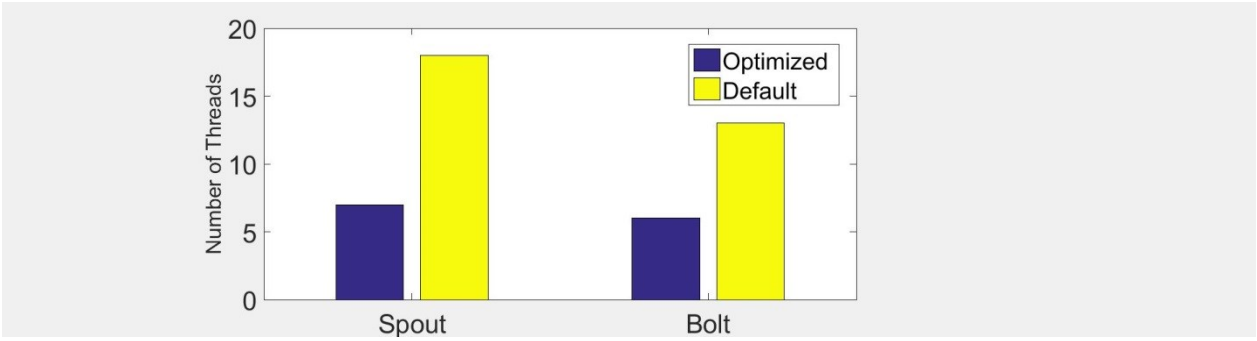

Figure 4.6: Number of Threads

Then, after applying the proposed algorithm, it would be able to minimize the number of threads. As shown in Figure 4.6, when the number of streams in the proposed scheduler increase, a better performance and the gap between the default and optimal resource allocation enhances. After running the experiment, it is validated that the proposed algorithm help to make the delay still follow the QoS constrain as shown in Figure 4.7.

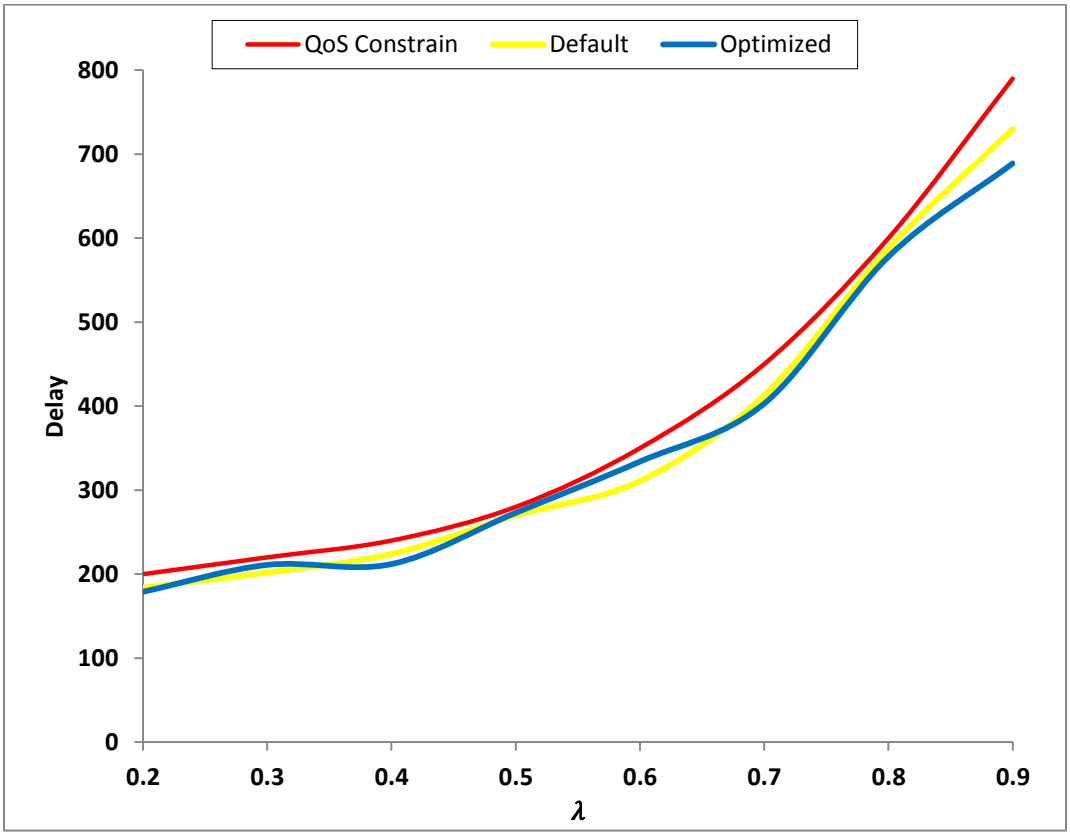

Figure 4.7: Measured Threads Processing Delay and Optimized Processing Delay

Chapter 5

Conclusions and Future Work

5.1 Conclusions

Cloud computing has already led a revolution in traditional Information Technology industry, it helps developers and companies overcome the lack of capacity in hardware (e.g. CPU and storage) and allows user indicates resources through the Internet in an on-demand fashion. Because of the rapid development rate in big data area, it would be valuable to do research in this area.

In this thesis, the importance of big data was addressed at first. And as the real time processing was not well studied as batch processing such as Hadoop/MapReduce, so we focus on the real time streaming processing platform Strom in this research.

First, the structure and data model of Storm was introduced and discussed in details. They are the basic fundament in studying the way how Storm works. Then, the modeling of Storm topology by grouping task strategies in parallel computing was addressed, with the task strategies model, it would be possible to calculate the process delay. And with the help of calculating the waiting time for each task in the topology by solving $G/M/c$ and $G/G/c$ queues in different situations, for satisfying the service requirement of the whole job, we are able to obtain the minimum number of

threads for each component. The proposed algorithm is able to capture the characteristics of the parallel computing, and the experimental result shows accuracy and could be applied in all kinds of stream processing topology structures. Finally, we proposed a heuristic algorithm to mitigate the complexity of the optimization problem.

The main contribution of this research is that we found a way to help the developers to determine the minimal parallelism hint in deploying the actual system and propose a model to capture the

5.2 Future Work

In this paper, we addressed the optimization problem by proposing a heuristic algorithm. With the help of the proposed algorithm, we are able to minimize the total number of in-service threads considering the response delay constraint of streams. However, the optimization problem which we solved values in general situation, which means that, in some specific situation, it may need more work in that area.

As it is mentioned before, there are a lot of works have already been done about big data. Researches are mainly about the scalability and fault tolerance as the main objectives for designing and building, which means that they will be pretty effective in helping researchers doing predictive and making the guild for the scientists in the future design. For future work, we would like to find the method to help the developers set the parallelism hint in an optimal way to provide the best performance in using Storm in processing real-time stream.

Bibliography

1. Armbrust, M., Fox, A., Griffith, R., Joseph, A.D., Katz, R.H., Konwinski, A., Lee, G., Patterson, D.A., Rabkin, A., Stoica, I., Zaharia,M.*: Above the clouds: a Berkeley view of cloud computing.* Technical Report No. UCB/EECS-2009–28, Electrical Engineering and Computer Sciences, University of California at Berkeley. (2009)
2. Q. Zhang, L. Cheng, R. Boutaba: *Cloud computing: state-of-the-art and research challenges.* Journal of Internet Services and Applications, 1. (2010)
3. Min Chen, Shiwen Mao, Yunhao Liu: *Big data: A survey.* Mobile Networks and Applications 19(2) · March 2014 (2014)
4. Thusoo, A., Sarma, J.S., Jain, N., Zheng, S., Chakka, P., Ning, Z., Antony, S., Hao, L., Murthy, R.: *Hive—a petabyte scale data warehouse using Hadoop.* In: Proceedings of IEEE 26th International Conference on Data Engineering (ICDE). (2010)
5. Manyika J, McKinsey Global Institute, Chui M, Brown B, Bughin J, Dobbs R, Roxburgh C, Byers AH: *Big data: the next frontier for innovation, competition, and productivity.* McKinsey Global Institute. (2011)
6. Laney D: *3-d data management: controlling data volume, velocity and variety.* META Group Research Note, 6 February. (2001)
7. "*What is Big Data?*" Villanova University. URL http://www.villanovau.com/resources/bi/what-is-big-data/#.V0OD4lVViko/. Accessed date: 23 May 2016. (2016)
8. Gantz J, Reinsel: *Extracting value from chaos*. IDC iView, pp 1–12. (2011)
9. Hu, Han; Wen, Yonggang; Chua, Tat-Seng; Li, Xuelong: "*Towards scalable systems for big data analytics: a technology tutorial*". IEEE Access 2: 652–687. (2014)

10. Cox M, Ellsworth D: *Managing big data for scientific visualization.* In: ACM Siggraph ’97 course #4 exploring gigabyte datasets in real-time: algorithms, data management, and time-critical design, August, 1997. (1997)
11. Labrinidis A, Jagadish HV: *Challenges and opportunities with big data*. Proc VLDB Endowment 5(12):2032–2033 (2012)
12. *Hadoop Wiki*. URL http://wiki.apache.org/hadoop/. Accessed date: 26 May 2016. (2016)
13. A. Toshniwal, S. Taneja, A. Shukla, K. Ramasamy, J. M. Patel, S. Kulkarni, J. Jackson, K. Gade, M. Fu, J. Donham, N. Bhagat, S. Mittal, and D. Ryaboy: *Storm@twitter.* In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, SIGMOD ’14, ACM. (2014)
14. *Storm, distributed and fault-tolerant real-time computing.* URL https://storm.apache.org/. Accessed date: 25 February 2015. (2015)
15. Ch. Shengbo, Y. Sun, D.Ulas. K.L. Huang, P. Sinha, G. Liang, X. Liu, and N. B. Shroff: *When Queueing Meets Coding: Optimal-Latency Data Retrieving Scheme in Storage Clouds.* IEEE INFOCOM, pp. 1042-1050, 2014. (2014)
16. G. Liang and U. Kozat: *FAST CLOUD: Pushing the Envelope on Delay Performance of Cloud Storage with Coding.* IEEE/ACM Trans. Networking, vol 22, no 6, pp. 2012-2025, Nov 2013. (2013)
17. S. L. Garfinkel: *An evaluation of Amazons grid computing services: EC2, S3 and SQS.* Harvard University, Tech. Rep., 2007. (2007)
18. Y. Lu, Q. Xie, G. Kliot, A. Geller, J. Larus, and A. Greenberg: *Join idlequeue: A novel load balancing algorithm for dynamically scalable web services.* Elsevier Performance Evaluation, vol 68, no 11, pp. 1056-1071, 2011.
19. J. C. Beard and R. D. Chamberlain: *Analysis of a simple approach to modeling performance for streaming data applications.* In Proc. of IEEE Int’l Symp. on Modelling, Analysis and Simulation of Computer and Telecommunication Systems. (2013)
20. B. Urgaonkar, G. Pacifici, P. Shenoy, M. Spreitzer, and A. Tantawi: *An analytical model for multi-tier internet services and its applications.* In Proceedings of SIGMETRICS, June 2005. (2005)
21. V. W. Mak and S. F. Lundstrom: *Predicting performance of parallel computations.* IEEE Transactions on Parallel and Distributed Systems, July 1990. (1990)

22. M. Hirzel, R. Soul, S. Schneider, B. Gedik, and R. Grimm: *A catalog of stream processing optimizations.* ACM Computing Surveys (CSUR), vol.46, no. 4, pp. 130-139, 2014. (2014)
23. S. Kamburugamuve, G. Fox, D. Leake , J. Qiu: *Survey of distributed stream processing for large stream sources.* Berkley Technical report, 2013. (2013)
24. Herodotou, H.: *Hadoop Performance Models.* Technical Report CS-2011-05. Computer Science Department, Duke University. (2011)
25. Bardhan S, Menasce D: *Queuing network models to predict the completion time of the map phase of MapReduce jobs.* Proceedings of the Computer Measurement Group International Conference, Las Vegas, NV, USA, 2012. (2012)
26. K. An and A. Gokhale: *Model-driven performance analysis and deployment planning for real-time stream processing.* Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '13), 2013. (2013)
27. *Apache thrift.* URL http://thrift.apache.org/. Accessed date: 03 Mar 2015. (2015)
28. *Apache Zookeeper.* URL http://zookeeper.apache.org/. Accessed date: 05 Mar 2015. (2015)
29. *Apache Kafak.* URL http://kafka.apache.org/. Accessed date: 05 Mar 2015. (2015)
30. *Kestrel: A simple, distributed message queue system*. URL http://robey.github.com/kestrel/. Accessed date: 05 Mar 2015. (2015)
31. *Understanding the Parallelism of a Storm Topology.* URL https://storm.apache.org/releases/1.0.0/Understanding-the-parallelism-of-a-Storm-topology.html/. Accessed date: 01 June 2015 (2015)
32. Ch. Shengbo, Y. Sun, D.Ulas. K.L. Huang, P. Sinha, G. Liang, X. Liu, and N. B. Shroff: *When Queueing Meets Coding: Optimal-Latency Data Retrieving Scheme in Storage Clouds.* IEEE INFOCOM 2014. (2014)
33. R. Jain: *The Art of Computer Systems Performance Analysis.* New York: Wiley, Inc. 1991. (1991)
34. De-Ron Liang, and S K. Tripathi: *On performance prediction of parallel computations with precedent constraints.* Parallel and Distributed Systems, IEEE Transactions on, vol 11, no.5, pp. 491-508, 2000. (2000)
35. S.Vakilinia, MM. Ali, and D. Qiu: *Modeling of the Resource Allocation in Cloud Computing Centers.* Computer Networks, vol. 9, no. 1, pp. 453-470, 2015. (2015)
36. L. Kleinrock: *Queuing Systems, vol. I.* John Wiley and Sons, 1976. (1976)

37. S. K. Bose: *An Introduction to Queueing Systems.* Kluwer Academic, The Rosen Publishing Group, Springer, 2002. (2002)
38. P.N.Yianilos: *Data structures and algorithms for nearest neighbor search in general metric spaces.* In SODA , vol. 93, no. 194, pp. 311-321, January 1993. (1993)
39. D.P. Bertsekas, R.G. Gallager, and P.Humblet: *Data networks (Vol. 2).* New Jersey: Prentice-Hall International, 1992. (1992)